第13学时　引用与结构

如果Perl是你使用的第一个编程语言，那么本学时将会使你感到颇有兴趣。在大多数编程语言中，你会发现一个概念，即一组数据实际上可以是对另一组数据的引用。有时这些引用称为指针（在pascal或C语言中），有时这种技术称为间接引用（在汇编语言中），而有些语言则根本没有指针的概念（在BASIC或Java中）。如果你以前从未使用过引用、指针或间接引用等概念，那么可能必须多次阅读本学时讲解的某些部分的内容，否则会感到混淆不清。

Perl也拥有这些特殊类型的值，不过在 Perl中，它们都称为引用。在 Perl中，引用可以用于许多目的，但在本学时中，你要学习的是如何使用引用来调用带有多个参数的复杂函数和如何创建复杂的数据类型，如列表的列表。

所谓引用，它非常类似老式图书馆中的卡片目录。目录中的每个索引卡指的是图书馆中的一本书。卡片可以指明这本书是什么类型的书（比如小说、非小说、参考书等），并指明这本书放在什么位置。有些卡片目录可能配有对同一本书的若干个引用，它们是不同种类的引用，并且甚至可以参见该目录中的其他卡片。

Perl的引用类似卡片目录，可以指向各组数据。引用能够知道它指向的是何种类别的数据（如标量、数组或哈希），也知道这些数据在什么地方。引用可以被拷贝，但不改变原始数据的任何东西。对于同一组数据，可以进行多次引用。实际上一个引用可以指向其他的引用。

请牢记下面这些要点，慢慢阅读下面几页内容，并且在我们介绍有关的问题时保持清醒的头脑：

- 引用的基本概念。
- 引用的常见结构。
- 运用所有这些概念而建立的一个简要代码例子。

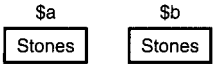
13.1　引用的基本概念

使用赋值运算符，可以创建和赋值一个普通的标量变量，如下所示：

```
$a="Stones";   # A normal scalar
```

在这个代码段建立后，可以创建一个称为 $a的标量变量，它包含字符串“ Stones ”。到现在为止，一切都很正常。这时，在计算机中的某个地方有一个标为 $a的位置，它包含了该字符串，如下图所示：

如果将标量$b赋予$a，比如$a=$b，那么会产生该数据的两个拷贝，它们使用两个不同的名字，如下图所示：



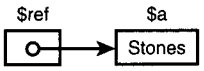
如果你想要两个独立的数据拷贝，那么拥有两个拷贝是很好的。但是，如果想让 $a和$b都引用同一组数据，而不是引用一个数据拷贝，那么必须创建一个引用。所谓引用，它只是

指向一组数据的指针，并不包含实际数据的本身。该引用通常存放在另一个标量变量中。

若要创建对某个既定变量的引用，可以在该变量的前面加上一个反斜杠。例如，若要创建称为$ref的对$a的引用，只需要像下面这样将引用赋予 $ref即可：

```
$ref=\$a;   # Create a reference to $a
```

这个赋值创建了类似下面这样的条件：



$ref并不包含用于它自己的任何数据，它只是对 $a的一个引用。变量$a根本没有改变，它仍然可以照常被赋值（$a=“ Foo ”）或显示(print $a)。

变量$ref现在包含对$a的引用。不能简单地对 $ref进行操作，因为它里边没有通常的标量值。实际上，如果输出 $ref，就会显示类似SCALAR(0x0000)的信息。若要通过 $ref获得$a中的值，必须间接引用 $ref。间接引用可以被视为上面的方块图中按箭头方向的引用。若要通过引用$ref来输出$a的值，你可以像下面这样使用另一个 $：

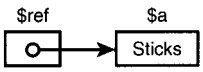
```
print $$ref;
```

在上面的代码段中， $ref当然包含了引用。增加的一个 $告诉Perl，$ref中的引用指的是一个标量值。$ref引用的标量值被取出并输出。

也可以通过引用来修改原始值，这是你对数据拷贝所不能进行的操作。下面这个代码用于修改$a中的原始值：

```
$$ref="Sticks";    # De-references $ref
```

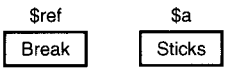
这项修改形成了类似下面这样的引用关系：



如果你使用$ref而不是$$ref

```
$ref="Break";
```

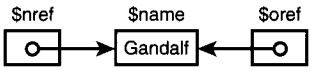
那么存放在$ref中的引用将被撤消并被实际值取代，如下所示：



上面这个代码段运行后， $ref不再包含一个引用，它只是一个标量。你可以像任何其他标量值那样，给引用赋值：

```
$name="Gandalf";
$nref=\$name;        # Has a reference to $name
$oref=$nref;         # Has a copy of the reference to $name
```

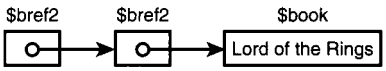
得到的结果如下：



上面的代码段运行后， $$oref和$$nref均可用于获取值“ Gandalf ”。也可以存放对某个引用的引用，如下所示：

```
$book="Lord of the Rings";
$bref=\$book;    # A reference to $book
$bref2=\$bref;   # A reference to $bref (not to $book!)
```

在这个例子中，引用链接类似下面的形式：



如果使用$bref2来输出书名，那么该引用将是$$bref2，如果使用$bref，则该引用是$$bref。请注意，$$$bref2多了一个美元符号，它需要增加一层间接引用，才能获得原始值。

13.1.1 对数组的引用

也可以创建对数组和哈希结构的引用。可以像创建对标量的引用那样，使用反斜杠来创建对数组和哈希结构的引用：

```
$aref=\@arr;
```

现在标量变量$aref包含了对整个数组@arr的引用。直观地说，它类似下面的形式：

若要使用引用$aref来访问@arr的各个部分，你可以使用下列代码之一：

```
$$aref[0]       @arr的第一个元素
@$aref[2, 3]    @arr的一个片
@$aref          @arr的整个数组
```

为了清楚起见，可以使用花括号将引用与涉及数组的各个部分隔开，如下所示：

```
$$aref[0]       与 ${$aref}[0]相同
$$aref[2, 3]    与 ${$aref}[2, 3]相同
@$aref          与 @{$aref}相同
```

例如，若要使用数组引用$aref，以便输出@arr的所有元素，可以使用下面这个代码：

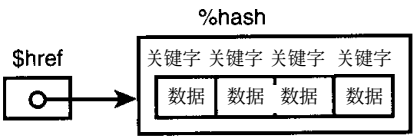
```
foreach $element (@{$aref}) {
    print $element;
}
```

13.1.2 对哈希结构的引用

若要创建对哈希结构的引用，可以使用反斜杠，就像创建标量和数组的引用那样：

```
$href=\%hash;
```

上面这个代码段用于创建对哈希结构 %hash的引用，并将它存放在 $href中。这个代码段创建的引用结构类似下面的形式：



若要使用对哈希的引用%href来访问%hash的各个部分，可以使用下面这些代码例子：

```
$$href{key}     访问%hash中的一个关键字，也可以是${$href}{key}
%$href          访问整个哈希结构，也可以是%{$href}
```

若要迭代通过该哈希结构，输出所有的值，可以使用下面这个代码：

```
foreach $key (keys %$href) {
    print $$href{$key};    # same as $hash{$key}
}
```

13.1.3 作为参数的引用

由于整个数组或哈希结构均可被引用，并且该引用可以存放在一个标量中，因此，借助这些引用，你可以调用带有多个数组或哈希结构的函数。

你可能还记得第8学时中我们讲过，下面这种代码段是不能运行的：

```
# Buggy!
sub getarrays {
    my(@a, @b)=@_;
    :
}
@fruit=qw(apples oranges banana);
@veggies=qw(carrot cabbage turnip);
getarrays(@fruit, @veggies);
```

这个代码不能运行，因为 getarrays(@fruit，@veggies)将两个数组压缩到单个数组 @_中。在getarrays()函数中，将@a和@b赋予@_，会导致现在存放在@_中的@fruits和@vegetables的所有元素都被赋予@a。

当所有数组挤入@_之后，就没有办法知道一个数组在何时结束和下一个数组在何时开始。只有一个很大的统一的列表。

这就是引用可以发挥作用的地方。你不必将整个数组传递给 getarrays，只要传递对这些数组的引用，就能够很好地达到你的目的：

```
# Works OK!
sub getarrays {
    my($fruit_ref, $veg_ref)=@_;
    :
}
@fruit=qw(apples oranges banana);
@veggies=qw(carrot cabbage turnip);
getarrays(\@fruit, \@veggies);
```

函数getarrays()总是接收两个值，即两个引用，无论这些引用指向的数组有多长。这时，$fruit_ref和$veg_ref可以用来显示或编辑数据，如下所示：

```
sub getarrays {
    my($fruit_ref, $veg_ref)=@_;

    print "Fruits:", join(',', @$fruit_ref);
    print "Veggies:", join(',', @$veg_ref);

}
```

当你将对标量、数组或哈希结构的引用作为参数传递给函数时，有几个问题必须记住。当你传递引用时，函数能够对引用指向的原始数据进行操作。请看下面这些例子：

```
# Passing Values

sub changehash {

    my(%local_hash)=@_;

    $local_hash{mammal}='bear';
```

```
# Passing references

sub changehash {

   my($href)=@_;

   $$href{mammal}='bear';
```

```
    return;
}
%hash=(fish => 'shark',
bird=> 'robin');
changehash(%hash);
```

```
    return;
}
%hash=(fish => 'shark',
bird=> 'robin');
changehash(\%hash);
```

在左边的例子中，当按正常情况传递哈希结构时， @_取得原始哈希结构%hash中每个关键字值对的各个值。在子例程 changehash()中，现在放入@_中的哈希结构的各个元素被拷贝到称为%local_hash的新哈希结构中。哈希%local_hash被修改，该子例程返回。当子例程返回后，%local_hash就被撤消，而程序的主要部分中的%hash则保持不变。

在右边这个例子中，对%hash的引用通过@_被传递到子例程changehash()中。该引用被拷贝到标量$href中，它仍然指原始哈希%hash。在子例程中， $href指向的哈希结构被修改，子例程返回。changehash()返回后，原始哈希结构%hash将包含新关键字bear。

当数组@_用于传递子例程参数时，它是个引用的数组。修改 @_数组的元素就会改变传递到函数中的原始值。修改传递给子例程的参数，通常被认为是不慎重的一种做法。如果你想让子例程修改传递给它们的参数，那么应该传递对子例程的引用。这种操作方法更加清楚。当传递一个引用时，可以认为原始值是可以修改的。

13.1.4 创建各种结构

创建对数组和哈希结构的引用，可以用来与子例程之间来回传递这些结构，并且可以用来创建下面我们很快就要介绍的一些复杂结构。不过你应该知道，当你创建了对哈希结构或数组的引用后，就不再需要原始哈希结构或数组。只要对哈希结构或数组的引用存在，即使原始数据不再存在， Perl仍然保留着哈希结构和数组的各个元素。

在下面的代码段中，代码块中创建了一个哈希结构 %hash，并且这个哈希结构是该代码块的专用结构：

```
my $href;
{
    my %hash=(phone=> 'Bell', light=> 'Edison');
    $href=\%hash;
}
print $$href{light};   # It will print "Edison"!
```

在这个代码块中，标量$href被赋予对%hash的引用。当该代码块存在时，即使 %hash已经消失， $href中的引用仍然有效（因为 %hash是代码块的专用结构）。当结构本身已经超出作用域之后，对该结构的引用仍然可以存在， $href引用的哈希结构仍然可以修改。

如果你观察上面这个代码块，就会发现，它的唯一目的是创建对哈希结构的引用。 Perl提供了一个机制，可以用来创建这样的引用，而不必使用中间的哈希结构 %hash。这个机制称为匿名存储。下面这个例子创建了一个对匿名哈希结构的引用，并把它存储在 $ahref中：

```
$ahref={ phone => 'Bell', light => 'Edison' };
```

花括号（{}）将哈希结构括起来，返回对它的引用，但实际上并没有创建新的变量。你

可以使用前面的“对哈希结构的引用”这一节中介绍的所有方法，对匿名哈希结构进行操作。

也可以使用方括号（[]）创建匿名数组。

```
$aaref=[ qw( Crosby Stills Nash Young ) ];
```

同样，也可以使用前面的“对数组的引用”这一节中介绍的方法对数组的引用进行操作。

当引用的变量本身超出作用域时（如果它是个专用变量），那么该引用指向的数据将全部消失，如下所示：

```
{
    my $ref;
    {
    $ref=[ qw ( oats peas beans barley ) ];
    }
    print $$ref[0];          # Prints "oats", $ref is still in scope
}
print $$ref[0];              # $ref is no longer in scope--this is an error.
```

如果use strict正在运行，那么上面这个代码段甚至不进行编译。 Perl将$ref的最后一个实例视为全局变量，这是不允许的。即使没有 use strict，Perl的-w警告特性也会输出一个undefined value（未定义的值）消息。

这些匿名哈希结构和匿名数组可以组合成某些结构形式，我们将在下一节中介绍这些结构。每个哈希结构和数组的引用代表一个标量值，并且由于它是单个标量值，因此可以存放在其他数组和哈希结构中，如下所示：

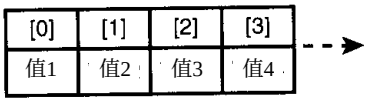
```
$a=[ qw( rock pop classical ) ];
$b=[ qw( mystery action drama ) ];
$c=[ qw( biography novel periodical )];

# A hash of references to arrays
%media=( music => $a, film => $b, 'print' =>$c);
```

13.2 结构的配置方法

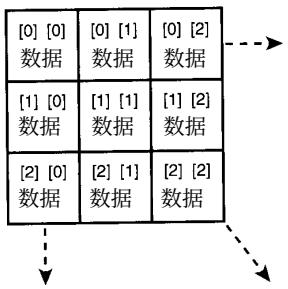
下面各节将介绍列表和哈希结构的一些常用结构配置方法。

13.2.1 一个例子：列表中的列表

在Perl中，列表中的列表常常用来代表一种称为二维数组的结构。也就是说，标准数组是个值的线性列表，如下所示：



二维数组类似一个值的表格，里面的每个元素按照轴上的一个点来进行编址。索引的第一部分表示行号（从0开始），第二部分是列号，请看下图：



Perl实际上并不支持真正的二维数组。 Perl允许你使用数组引用的数组，模仿建立二维数组。

若要创建数组的数组，请使用下面这个原义表达式：

```
@list_of_lists=(
    [ qw( Mustang Bronco Ranger ) ],
    [ qw( Cavalier Suburban Buick ) ],
    [ qw( LeBaron Ram ) ],
);
```

请认真观察上面的代码段。它创建了一个正则列表 @list_of_lists，但是它由对其他列表的引用所组成。若要访问最里层的列表的各个元素（即二维数组中的单元格），可以使用下面这个代码：

```
$list_of_lists[0][1];     # Bronco.  1st row, 2nd entry
$list_of_lists[1][2];     # Buick.   2nd row, 3rd entry
```

若要确定最外层的列表中的元素数目，你可以像对其他任何数组那样进行操作，使用 $# 表示法或者使用标量上下文中的数组名：

```
$#list_of_lists;          # Last element of @list_of_lists: 2
scalar(@list_of_lists);   # Number of rows in @list_of_lists: 3
```

若要确定里层列表中的某个列表的元素数目，可能有一点儿麻烦。语句 $list_of_lists[1]返回@list_of_lists的第二行中的引用。如果将它输出，则显示类似ARRAY（0x00000）这个数据。若要将@list_of_lists的一个元素当作数组来处理，请在它的前面加上一个符号 @，如下所示：

```
scalar(@{$list_of_lists[2]});    # From the 3rd row, 2 elements
$#{$list_of_lists[1]};           # From the 2nd row, last element is 2
```

若要遍历列表的列表中的每个元素，可以使用下面这个代码：

```
foreach my $outer (@list_of_lists) {
    foreach my $inner (@{$outer}) {
        print "$inner ";
    }
    print "\n";
}
```

可以添加下面这样的结构：

```
push(@list_of_lists, [ qw( Mercedes BMW Lexus ) ]);  # A new row
push(@{$list_of_lists[0]}, qw( Taurus ) );           # A new element to one list
```

13.2.2 其他结构

在上一节中，我们介绍了如何使用引用和数组创建基本的 Perl结构，即列表的列表。实际上可以将数量不受限制的数组、标量和哈希结构的变形组合起来，创建更为复杂的数据结构，比如下面这些结构：

- 哈希结构的列表。
- 列表的哈希结构。
- 哈希结构的哈希结构。
- 包含列表的哈希结构，而列表中又包含哈希结构，等等。

由于本书篇幅有限，无法一一介绍所有这些结构。你安装的每个 Perl所配备的在线文档包含了一个称为“ Perl Data Structures Cookbook(Perl的数据结构大全) ”文档。它详细而明白地描述了这些结构和许多其他数据结构。对于每种数据结构，“ Perl Data Structures Cookbook ”文档详细描述了下列信息：

- 说明你的结构（原义表示法）。
- 填充你的结构。
- 添加各个元素。
- 访问各个元素。
- 遍历整个数据结构。

若要查看“ Perl Data Structures Cookbook ”，请在命令提示符处键入 perldoc perldsc。

13.2.3 使用引用来调试程序

当使用引用对程序进行调试时，编程新手常常搞不清楚哪些引用指向什么种类的数据结构。另外，在你习惯之前，语句也容易混淆。Perl提供了一些工具，可以帮助你确定有关的情况。

首先，可以输出该引用。Perl能够显示该引用指向什么结构。例如，下面这个代码行：

```
print $mystery_reference;
```

可以显示

```
ARRAY(0x1231920)
```

这个结构意味着变量 $mystery_reference是对一个数组的引用。此外，变量也可以是对标量（SCALAR）、哈希结构（ HASH）或子例程（ CODE）的引用。若要输出 $mystery_reference指向的数组，可以将它作为数组来处理，如下所示：

```
print join(',',  @{$mystery_reference});
```

Perl的调试程序也配有一些程序工具，帮助你确定某个引用指向什么数据结构。在调试程序中，你可以像通常那样输出引用。下面这个代码段显示了一个被查看的名叫 $ref的引用：

```
DB<1> print $ref
HASH(0x20114dac)
```

显然，$ref是指一个哈希结构。该调试程序包括一个命令，即命令 x，它将输出该引用和它的内部结构：

```
   DB<2> x $ref
0  HASH(0x20114dac)
   'fruit' => 'grape'
   'vegetable' => 'bean'
```

在这个代码中，该引用包含一个带有两个元素（关键字‘ fruit ’和‘ vegetable ’）的哈希结构。该调试程序甚至能够输出列表的列表之类的复杂数据结构，如下所示：

```
   DB<1> x $a
0  ARRAY(0x20170bd4)
   0  ARRAY(0x20115484)     <-- First row in a list of lists
      0  5
      1  6                  <-- Elements in the first row
      2  7
   1  ARRAY(0x2011fbb4)     <-- Second row in the list of lists
      0  9
      1  10
      2  11
   2  ARRAY(0x2011faa0)     <-- Third row in a list of lists
      0  'a'
      1  'b'
      2  'c'
```

上面的例子显示了一个引用 $a，它指向一个数组 ARRAY（0x20170bd4）。而这个数组又包含3个别的数据引用，即 ARRAY（0x20115484）、ARRAY（0x2011fbb4）和 ARRAY

（0x2011faa0），每个数组包含3个元素。

模块Data∷Dumper包含的一些函数能够显示各个引用的内容。Data∷Dumper是独一无二的，它的输出格式是有效的Perl格式，它可以存入文件，并在以后被检索，以提供可存储的结构。Data∷Dumper模块将在第14学时中介绍。

13.3 练习：另一个游戏——迷宫

当你学习了那么多的新奇概念（引用和结构）之后，需要来一点消遣娱乐了。下面这个练习展示了一种结构和几个引用，并且你可以做一个简单的游戏。

采用探险和狩猎之类的传统游戏方式，你被置于一个迷宫之中，必须找到你的出路。这个迷宫并无奇特之处，它只是由一些房间所组成，并且每个房间至少有一个门。门可以通向位于东、南、西、北的相邻房间。这个游戏的目的是找到一间密室。你会发现通往该密室只有两条路，另外还有许多走不通的路。

首先，键入程序清单13-2，并将它保存为Maze。运行该程序，得到类似程序清单13-1的输出。

程序清单13-1 Maze的输出示例

```
1:  You may move East (e)
2:  Which way? e
3:  You may move South West East (swe)
4:  Which way? e
5:  :
6:  Which way? e
7:  You made it through the maze!
```

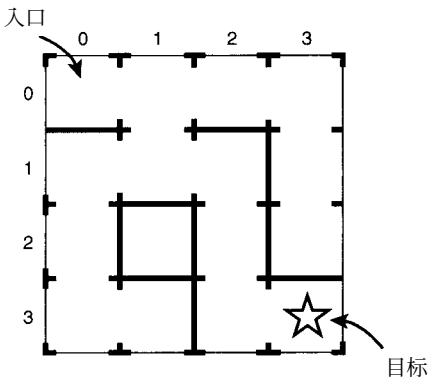
程序清单13-2 Maze的完整程序清单

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my @maze=(
5:        [ qw( e   swe we ws  ) ],
6:        [ qw( ne new sw ns ) ],
7:        [ qw( ns  -      ns wn  ) ],
8:        [ qw( ne w     ne w   ) ],
9:  );
10: my %direction=( n=> [ -1, 0], s=> [1,  0],
11:             e=> [  0, 1], w=> [0, -1]);
12:
13: my %full=( e => 'East', n => 'North', w=>'West', s=>'South');
14: my($curr_x, $curr_y, $x, $y)=(0,0,3,3);
15: my $move;
16:
17: sub disp_location {
18:     my($cx, $cy)=@_;
19:     print "You may move ";
20:     while($maze[$cx][$cy]=~/([nsew])/g) {
21:             print "$full{$1} ";
22:     }
23:     print "($maze[$cx][$cy])\n";
24: }
25: sub move_to {
```

```
26:       my($new, $xref, $yref)=@_;
27:
28:       $new=substr(lc($new),0,1);
29:       if ($maze[$$xref][$$yref]!~/$new/) {
30:               print "Invalid direction, $new.\n";
31:               return;
32:       }
33:       $$xref += $direction{$new}[0];
34:       $$yref += $direction{$new}[1];
35:  }
36:
37:  until ( $curr_x == $x and $curr_y == $y ) {
38:       disp_location($curr_x, $curr_y);
39:       print "Which way? ";
40:       $move=<STDIN>; chomp $move;
41:       exit if ($move=~/^q/);
42:       move_to($move, \$curr_x, \$curr_y);
43:  }
44:
45:  print "You made it through the maze!\n";
```

第1~2行：这两行代码是Perl程序正常的开始。-w使警告特性被激活， use strict用于捕获错误和不恰当的编程做法。

第4~9行：用于定义描述迷宫@maze的结构。显示的迷宫是个4×4的栅格，用一个列表的列表来表示。列表的每个元素用于描述迷宫中的任何一个房间可以通往哪些房间，因此，如果你重新设计这个迷宫，请务必留出一条出路。当前的迷宫如下所示：



有一个房间（2，1）是无法进入的，在这个结构中用一个 - 来表示这个房间。实际上，不能与n、s、e或w匹配的任何字符串均可使用。

第10~11行：当游戏的玩主向北或向南移动时，迷宫中的当前位置就需变更。哈希结构%direction用来根据老的位置和移动方向计算玩主的新位置。如果向“北”移动，则使玩主的x坐标移动-1（向上），y坐标保持不变。如果向“东”移动，则玩主的x坐标不变，而y坐标增加1。你将在第33~34行代码中看到坐标的变更情况。

第13~15行：程序中使用的变量用 my进行声明，以便使 use strict恰当地运行。存放在$curr_x和$curr_y中的玩主当前位置被设置为0，0。最后目的地$x和$y被设置为3，3。

第17行：根据栅格中的x，y坐标，该函数显示玩主可以在每个房间中移动的方向。

第20行：在$maze[$cx][$cy]的房间描述中选择字母n、s、e和w，每次选择1个字母。从哈希结构%full中显示nsew方向的相应描述。这个哈希结构只用于将短名字 (n)转换成长名字

(Norfh)，供显示之用。

第25行：该函数取出一个方向（存放在$new中）和对玩主的当前位置的引用。

第28行：方向用lc改为小写字母，substr只取出第一个字母，并将它赋予$new。这样，East变为e，West变为w，s仍为s。

第29行：搜索当前房间的$maze[$$xref][$$yref]，找出给定的方向（n、s、e和w）。如果不存在给定的方向，那么它对该房间无效，然后输出一条消息。

第33~34行：玩主的x和y坐标被更改。如果方向是e，则$direction{e}是对两个元素的数组的引用（0，1）。x坐标将递增0，即$direction{e}[0]。Y坐标将递增1，即$direction{e}[1]。

第37行：程序的主体从这里启动运行。该循环将不断运行，直到玩主的x和y坐标（$curr_x，$curr_y）与密室的坐标（$x，$y）相一致为止。

第38行：显示当前房间的“映像”。

第39行：需要的移动方向读入$move，用chomp删除换行符。如果玩主键入以q开头的任何信息，则游戏结束。

第42行：根据玩主当前需要做的移动和对玩主坐标的引用，调用子例程move_to()。move_to()子例程通过调整$curr_x和$curr_y，使玩主作相应的移动。

若要修改迷宫，使之采用另一种布局，只需改变存放在@maze中的栅格。迷宫不一定需要做成正方形，也不需要给每个房间制作映像，甚至不需要存在一条有效的路径。不过请记住，迷宫不要从它边上的某个房间开始。程序不会检查迷宫的有效性，不过，如果你创建了一个无效迷宫，Perl就会发出警告。如果要移动迷宫中的密室，只需改变它的$x和$y的值。

13.4 课时小结

本学时我们介绍了引用的基本概念。首先，讲述了如何创建对Perl的基本数据结构——标量、数组和哈希结构的引用。然后，介绍了如何使用这些引用，对原始数据结构进行操作。接着，说明了如何创建对哈希结构或数组的引用，不过这种引用没有与此相关的变量名，这种引用称为匿名存储。最后介绍了如何使用引用来创建复杂的数据结构，以及何处可以查找已有数据结构的文档资料。

13.5 课外作业

13.5.1 专家答疑

问题：当我用print“@LOL”输出一个列表的列表时，它输出的是ARRAY（0x101210），ARRAY（0x101400）等等，为什么？

解答：对于正规数组来说，print“@array”将输出数组的元素，各个元素之间有一个空格。Print“@LOL”也产生这样的结果，输出@LOL中的各个数组元素。若要输出@LOL中每个数组的组件，你必须使用本学时开头的“一个例子：列表中的列表”这一节中介绍的方法。

问题：我试图使用$ref=\($a，$b，$c)创建一个对列表的引用，结果却产生了一个对标量值而不是列表的引用，为什么？

解答：在Perl中，\（$a，$b，$c）实际上是（\$a，\$b，\$c）的简化形式。你得到的结果实际上是对括号中最后一个元素$c的引用。若要获得一个对匿名数组的引用，你应该使用

$ref=[$a，$b，$c]。

13.5.2 思考题

1) 语句$ref=\“ peanuts ”；运行后，$ref中包含了什么？

a. 什么也不包含。该语句无效。

b. peanuts。

c. 对一个匿名标量的引用。

2) 下面这个结构可以创建什么？

```
$a=[
    { name=> "Rose", kids=> [ qw( Ted, Bobby, John ) ] },
    { name=> "Marge", kids=>[ qw( Maggie, Lisa, Bart ) ] },
];
```

a. 一个哈希结构的哈希结构，它包含一个列表

b. 一个哈希结构的列表，它包含一个列表

c. 一个列表的列表，它包含另一个列表

13.5.3 解答

1) 答案是c。你可以创建对任何值的引用，而不只是创建对标量、数组和哈希变量的引用。你也可以用$ref=\100；创建对一个数字的引用。如果你的答案是 a，那么最好在一个短程序或调试程序中试用一些新数据，看看它们能够产生什么结果。

2) 答案是b。在本学时中我们没有具体介绍这种结构，不过你应该能够猜到这是个什么结构。一个哈希结构（花括号中）的列表（外层方括号）包含一个列表（ kids的数据）。

13.5.4 实习

- 修改Maze游戏，使之也能按对角线方向移动。你可以使用 4个新关键字来表示这些方向（ne、nw、se和sw是很难编程的）。提示：关键是修改@maze时使用新符号和%direction来表示按什么方向移动，比如[1，1][-1，-1]等。
- 设计一种结构（即使是纸上谈兵也行），来描述一种电话帐单。帐单本身包含类似哈希结构的关键字和数据（名字，电话号码，地址），帐单的某些部分是列表（明细电话项）。每个明细电话项也可以被视为一个哈希结构（电话接收方，时间）。